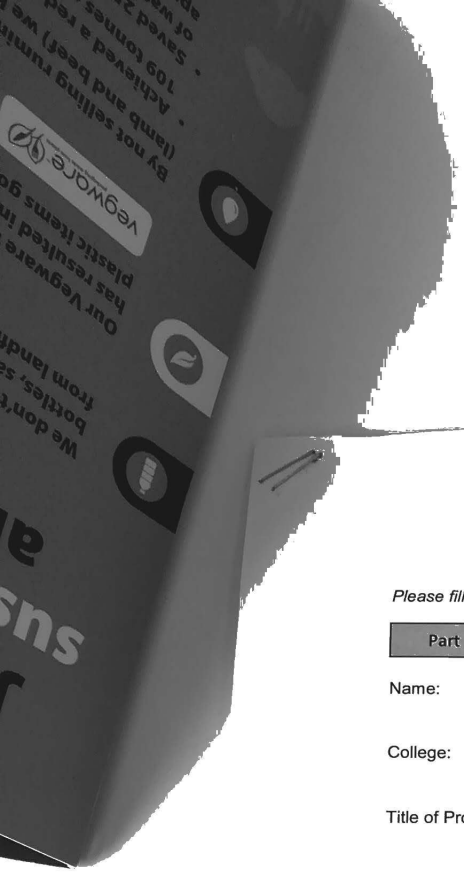

Computer Science Tripos

Part II Project Proposal Coversheet

Please fill in Part 1 of this form and attach it to the front of your Project Proposal.

Part 1

Name: Phoebe Nichols CRSID: pmn29

College: Churchill College Overseers: (Initials) amp12, rkm38

Title of Project: An Implementation of Prolog

Date of submission: 19/10/2018 Will Human Participants be used? No

Project Originator: Phoebe Nichols Signature: Phoebe Nichols

Project Supervisor: Prof. Alan Mycroft Signature: A Mycroft

Directors of Studies: Dr John Fawcett Signature: John Fawcett

Special Resource Sponsor: Signature: ----------

Special Resource Sponsor: Signature: ----------

Above signatures to be obtained by the Student

Part 2

Overseer Signature 1: ----------

Overseer Signature 2: ----------

Overseers signatures to be obtained by Student Administration.

Overseers Notes:

Part 3

SA Date Received: SA Signature Approved:

Project Originator:	Phoebe Nichols	**Project Supervisor:**	Prof. Alan Mycroft
Director of Studies:	Dr John Fawcett	**Overseers:**	Prof. Andrew Pitts Dr Rafal Mantiuk

Introduction and Description of the Work

Prolog is a declarative programming language. This means that a Prolog programmer decla rules that describe the result of a program, but does not specify how the program should achi this result. This is in contrast to imperative programming languages such as Java, where t programmer specifies how a program's state should be changed in order to achieve the resu of the program. The use of a declarative language presents the interesting problem of how evaluate a query without being told how the answer to the query should be computed. In t case of Prolog, this problem is solved by performing a graph search over the Horn clauses give in a Prolog program. This search attempts to find an assignment of values to variables tha satisfies the query by repeatedly unifying a literal with the head of a clause and then trying t prove that clause. and backtracking if no more progress can be made. The query is rejected if n assignment to satisfy the query exists. Typically, a left-to-right depth-first search is performed

The aim of this project is to implement a compiler and abstract machine for Prolog. Th target for the Prolog compiler will be specialised byte-code. Designing this byte-code will form part of the project. The abstract machine implemented will interpret this byte-code to run the original Prolog program. An existing specification of an abstract machine to execute Prolog the Warren Abstract Machine (WAM), will be used as inspiration—but I will not implement the full WAM.

The project will be restricted to a basic subset of Prolog so that it remains feasible. Some extensions that I am considering (for example type checking) will require an extension to Prolog syntax in order to provide information about program properties. The syntax I use may therefore not be a true subset of standard Prolog.

Starting Point

Prolog as a programming language

Computer Science Tripos: Part II Project Proposal

An Implementation of Prolog

Phoebe Nichols, pmn29
Churchill College
October 2018

Project Originator:	Phoebe Nichols	**Project Supervisor:**	Prof. Alan Mycroft
Director of Studies:	Dr John Fawcett	**Overseers:**	Prof. Andrew Pitts Dr Rafal Mantiuk

Introduction and Description of the Work

Prolog is a declarative programming language. This means that a Prolog programmer declares rules that describe the result of a program, but does not specify how the program should achieve this result. This is in contrast to imperative programming languages such as Java, where the programmer specifies how a program's state should be changed in order to achieve the result of the program. The use of a declarative language presents the interesting problem of how to evaluate a query without being told how the answer to the query should be computed. In the case of Prolog, this problem is solved by performing a graph search over the Horn clauses given in a Prolog program. This search attempts to find an assignment of values to variables that satisfies the query by repeatedly unifying a literal with the head of a clause and then trying to prove that clause, and backtracking if no more progress can be made. The query is rejected if no assignment to satisfy the query exists. Typically, a left-to-right depth-first search is performed.

The aim of this project is to implement a compiler and abstract machine for Prolog. The target for the Prolog compiler will be specialised byte-code. Designing this byte-code will form part of the project. The abstract machine implemented will interpret this byte-code to run the original Prolog program. An existing specification of an abstract machine to execute Prolog, the Warren Abstract Machine (WAM), will be used as inspiration—but I will not implement the full WAM.

The project will be restricted to a basic subset of Prolog so that it remains feasible. Some extensions that I am considering (for example type checking) will require an extension to Prolog syntax in order to provide information about program properties. The syntax I use may therefore not be a true subset of standard Prolog.

Starting Point

Prolog as a programming language

I will need a good understanding of how to program in Prolog in order to implement the Prolog compiler. I will also need to write test programs for the subset of Prolog that I have chosen to compile. These tests will help me to evaluate the correctness and performance of my compiler and abstract machine implementation.

I am studying the 50% course, which does not cover Prolog until Lent term this year. I have, therefore, studied the Prolog course over the summer.

Prolog compilation and execution

I have no prior experience with implementing Prolog interpreters or compilers. The IB 75%/II 50% Prolog course that I studied over the summer introduced me to the basics of Prolog

execution. I have also read around this area over the summer. Some of
cited in this document. I have not previously implemented any part of a Prol
interpreter, or abstract machine.

OCaml

I am intending to implement all of my project in OCaml initially, although I may add
Prolog for extension features. I have chosen OCaml because I have experience in
lexers and parsers in ML from the compilers course, and found ML convenient for
thing. OCaml will have the same advantages as ML, but is more widely used and th
more tools and libraries available.

I have very little prior experience with OCaml (only from the IB compilers course
need to dedicate time to studying it. This time is provisioned in my project plan.

Structure of the Project

The project is made up of four main components:

1. Lexer
2. Parser
3. Translator
4. Abstract machine

I will implement the components in this order. This is because each component ca
to generate test cases for the next component. As an extension, I may add a semanti
phase acting on the parse tree. This would verify program properties and try t
information so that the translator can generate more efficient code.

Implementation of the Abstract Machine

The abstract machine will be inspired by the Warren Abstract Machine (WAM) [1]. T
instruction set contains five main types of instruction:

- *Get* instructions, used to fetch a procedure's arguments. A procedure is a set
  with the same arity and function symbol.
- *Put* instructions, used to provide the arguments to a procedure.
- *Unify* instructions, used to perform unification.
- *Procedural* instructions, used for control transfer and environment allocation for p
  calls.
- *Indexing* instructions, used to link the different clauses making up a procedure.

These are interpreted using three main code areas called the local stack, heap (global sta
trail. The local stack contains environments and choice points (information needed f
tracking), the heap contains structures and lists created by unification, and the trail
references to variables that were bound on unification and need to be unbound on backt

The WAM is the most standard Prolog implementation, and is used by SWI-Prolog
relevant literature includes: a paper describing a Prolog abstract machine that av
register usage of the WAM [2], and a general comparison of Prolog implementation tec
[3]. A lower-level adaptation of the WAM is given by Peter Van Roy, achieving in
performance [4]. Since all these papers base themselves on the WAM, I will be able
rough WAM implementation using some other ideas taken from the literature.

Testing

Unit tests will be written for each individual component. Components written in OCa
likely have unit tests written with the testing framework OUnit [5]. Any components

execution. I have also read around this area over the summer: some of the papers I read are cited in this document. I have not previously implemented any part of a Prolog compiler, interpreter, or abstract machine.

OCaml

I am intending to implement all of my project in OCaml initially, although I may add some SWI-Prolog for extension features. I have chosen OCaml because I have experience implementing lexers and parsers in ML from the compilers course, and found ML convenient for this sort of thing. OCaml will have the same advantages as ML, but is more widely used and therefore has more tools and libraries available.

I have very little prior experience with OCaml (only from the IB compilers course), so I will need to dedicate time to studying it. This time is provisioned in my project plan.

Structure of the Project

The project is made up of four main components:

1. Lexer
2. Parser
3. Translator
4. Abstract machine

I will implement the components in this order. This is because each component can be used to generate test cases for the next component. As an extension, I may add a semantic analysis phase acting on the parse tree. This would verify program properties and try to provide information so that the translator can generate more efficient code.

Implementation of the Abstract Machine

The abstract machine will be inspired by the Warren Abstract Machine (WAM) [1]. The WAM instruction set contains five main types of instruction:

- *Get* instructions, used to fetch a procedure's arguments. A procedure is a set of clauses with the same arity and function symbol.
- *Put* instructions, used to provide the arguments to a procedure.
- *Unify* instructions, used to perform unification.
- *Procedural* instructions, used for control transfer and environment allocation for procedure calls.
- *Indexing* instructions, used to link the different clauses making up a procedure.

These are interpreted using three main code areas called the local stack, heap (global stack), and trail. The local stack contains environments and choice points (information needed for backtracking), the heap contains structures and lists created by unification, and the trail contains references to variables that were bound on unification and need to be unbound on backtracking.

The WAM is the most standard Prolog implementation, and is used by SWI-Prolog. Other relevant literature includes: a paper describing a Prolog abstract machine that avoids the register usage of the WAM [2], and a general comparison of Prolog implementation techniques [3]. A lower-level adaptation of the WAM is given by Peter Van Roy, achieving improved performance [4]. Since all these papers base themselves on the WAM, I will be able to do a rough WAM implementation using some other ideas taken from the literature.

Testing

Unit tests will be written for each individual component. Components written in OCaml will likely have unit tests written with the testing framework OUnit [5]. Any components written

in SWI-Prolog will be tested using native support for unit tests as described in the SWI-Prolog documentation [6]. I will write unit tests for each component along with the implementation of the component itself.

Evaluation

I will be able to quantitatively evaluate the project in terms of:

- The run-time performance of my abstract machine compared to standard Prolog implementations such as SWI-Prolog.
- The run-time performance of my abstract machine compared to my interpreter.
- The change in code size, number of choice points, or performance, for any optimisations that I implement.

Here, 'performance' means a comparison of space and time usage for sample programs. I will be able to profile the memory behaviour of my OCaml programs using Spacetime [7]. This is a tool built into special versions of the OCaml compiler for memory profiling. SWI-Prolog also includes predicates for execution profiling [8].

Success Criterion

My success criterion is to implement the following components for a basic subset of Prolog:

- Lexer
- Parser
- Translator from parse tree to byte-code
- Abstract machine to execute this byte-code

These components should enable me to correctly execute example programs written in the grammar that I use. The components will not implement the full ISO Prolog standard, and the grammar used may include features such as type annotations not found in standard Prolog. The lexer and parser may be implemented manually or using tools such as ocamllex, ocamlyacc, or menhir.

Possible extensions

The main extensions for this project involve adding optimisations to the compiler. Some potential optimisations are:

- **Last call optimisation**
  Tail recursion can be converted to iteration to save adding a new stack frame for each tail recursive call.
- **Determinacy analysis**
  A clause is said to be determinate if it can only return one possible solution [9]. This information can be used to avoid backtracking through determinate clauses.
- **Mode analysis**
  The argument to a predicate can be assigned a mode to say that the argument is always used as input, or as output [10]. Mode information can be used to invoke special purpose unification routines that test for fewer cases and therefore can be faster.
- **Type checking**
  It is possible to define a polymorphic type system for Prolog [11]. Type information can be used to select faster, special-purpose unification routines. A type checker also has the significant advantage of potentially spotting program bugs and saving developer time.

There are many more possible optimisations I could apply, and I might also select one of these for an extension. As I implement the initial system I will learn more about the optimisations I could make, and may also find my implementation to be more suited to certain optimisations.

Timetable and Milestones

Key summary

- 16th Dec: Success criteria met
- 1st Feb: Progress report deadline
- 24th Feb: Draft dissertation completed
- 17th May: Dissertation deadline

***19th Oct**: Proposal submitted*

22nd Oct – 4th Nov

Project set-up:

- Investigate and set up an IDE for OCaml.
- Set up a backup system for the project.
- Choose a software engineering methodology to follow.
- Learn to use the testing frameworks needed for the project.

Write small programs in SWI-Prolog, to re-enforce understanding of Prolog and for potential use as benchmarks. Study OCaml using the Real World OCaml textbook [12], and write small programs in OCaml to gain familiarity with the language.

Milestones:

- Document produced to describe the development environment, backup system, software engineering methodology, and testing frameworks used.
- Example programs written in OCaml and Prolog.

5th Nov – 18th Nov

Implement a lexer and parser for the chosen subset of Prolog. I am intending to use ocamllex and ocamlyacc (described in [13]) for this.

Milestones:

- Code for lexer written and tested.
- Code for parser written and tested.
- Document produced to describe the implementation of these components.

***30th Nov**: Last day of Michaelmas term*

19th Nov – 2nd Dec

Implement an interpreter to evaluate the parse tree output by the parser. This interpreter will guide the design of the abstract machine, and may be useful as a performance benchmark.

Milestones:

- Code for interpreter written and tested.
- Document produced to describe the implementation of the interpreter.

3rd Dec – 16th Dec

Design the byte-code to represent Prolog programs, and write an abstract machine to execute this byte-code. Write the translator to convert the parse tree to byte-code.

Milestones:

- Code for abstract machine written and tested.

- Code for translator written and tested.
- Document produced to describe the byte-code format chosen.
- Document produced to describe how the abstract machine works.

17th Dec – 23rd Dec [*Slack*]

Read about possible extensions to the project. This could be research on type, mode, or determinacy analysis for Prolog. Of these options I expect type checking to be the most complex, and determinacy analysis the least complex. I will choose where to invest my time depending on how far ahead or behind the project is.

Milestones:

- Produce document to summarise research into optimisations.

24th Dec – 30th Dec [*Christmas break*]

31st Dec – 6th Jan [*Slack*]

Continue research into extensions.

Milestones:

- Add more content to the previous summary document.

***15th Jan**: First day of Lent term*

7th Jan – 20th Jan [*Slack*]

Implement extension features for the compiler.

Milestones:

- Code for extension features written and tested.
- Document produced to describe the theory and implementation of the extension features.

21st Jan – 27th Jan

Prepare for progress report and presentation.

Milestones:

- Progress report and presentation written.

***1st Feb**: Progress report deadline*

28th Jan – 10th Feb

Write a draft of the structure of the dissertation, giving the main points to be covered. Write a full version of the evaluation section of the dissertation.

Milestones:

- Draft structure of dissertation written and sent to DoS and supervisor.
- Evaluation section written and sent to DoS and supervisor.

11th Feb – 24th Feb

Respond to any feedback on the draft structure or evaluation section.

Write the introduction, preparation, implementation, and conclusion sections of the dissertation.

Milestones:

- Dissertation completed and sent to DoS and supervisor.

25th Feb – 10th Mar [*Slack*]

I will use this time to finish the dissertation if the first draft is not already finished. If a draft

is finished then I will likely be waiting for feedback and will dedicate this time to revision.

Milestones:

- If the previous milestone was missed then it should now be met.

11th Mar – 24th Mar *15th Mar: Last day of Lent term*

Respond to any feedback on the dissertation.

Milestones:

- Dissertation updated following any feedback received.

25th Mar – 21st Apr [*Slack*]

Continue improving the dissertation and responding to feedback.

Milestones:

- Dissertation updated.

22nd Apr – 5th May *23rd Apr: First day of Easter term*

Make any final changes and submit the dissertation.

Milestones:

- Dissertation submitted. *17th May: Dissertation deadline*

Resources Required

I am intending to use my own laptop for the project (Dell XPS 15, 16GB RAM, i7-7700HQ, running Ubuntu). I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure.

The basic resources that are essential to my project are the OCaml compiler `ocamlopt` and the SWI-Prolog interpreter `swipl`. These are both installed on the MCS system, so I will be able to complete the project using the MCS machines if my laptop fails.

I will use git for revision control. My git repository will be hosted in GitHub, and this will serve as a cloud backup of my code. I will store my local copy of the git repository inside a Dropbox folder so that my work is also all automatically synced to Dropbox. I will perform weekly backups to a USB stick in case these other backup techniques fail.

References

[1] David H. D. Warren. An abstract Prolog instruction set. *SRI international*, 1983.

[2] Neng-Fa Zhou. A register-free abstract prolog machine with jumbo instructions. In Véronica Dahl and Ilkka Niemelä, editors, *Logic Programming*, pages 455–457, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[3] Andreas Krall. Implementation techniques for prolog. *Institut für Computersprachen Technische Universität Wien*, 1985.

[4] Peter Lodewijk Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming?* PhD thesis, 1990.

[5] OUnit. `hhttp://ounit.forge.ocamlcore.org/`. OCaml unit testing framework.

[6] Jan Wielemaker. Prolog Unit Tests. `http://www.swi-prolog.org/pldoc/doc_for?object=section(%27packages/plunit.html%27)`. SWI-Prolog Documentation.

[7] Memory profiling with Spacetime. `https://caml.inria.fr/pub/docs/manual-ocaml/spacetime.html`. OCaml Documentation.

[8] Execution profiling. `http://www.swi-prolog.org/pldoc/man?section=profile`. SWI-Prolog Documentation.

[9] Thomas W. Getzinger. The costs and benefits of abstract interpretation-driven prolog optimization. In *Proceedings of the First International Static Analysis Symposium on Static Analysis*, 1994.

[10] Saumya K. Debray and David S. Warren. Automatic mode inference for logic programs. *The Journal of Logic Programming*, 5(3):207 – 229, 1988.

[11] Alan Mycroft and Richard A. O'Keefe. A polymorphic type system for prolog. *Artificial Intelligence*, 23(3):295 – 307, 1984.

[12] Jason Hickey Anil Madhavapeddy and Yaron Minsky. *Real World OCaml: Functional Programming for the Masses*. O'Reilly Media.

[13] Lexer and parser generators (ocamllex, ocamlyacc). `https://caml.inria.fr/pub/docs/manual-ocaml/lexyacc.html`. OCaml Documentation.